Learning from Failure: Inference-Time Self-Improvement for Computer-Use Agents

Xueqiao Sun[1,2], Xiaohan Wang[1], Ludwig Schmidt[1], Serena Yeung-Levy[1,†], and Yuhui Zhang[1,†]

[1]Stanford University, Stanford, CA 94305, USA
[2]Tsinghua University, Beijing, 100084, China

Abstract. Computer-use agents, which leverage multimodal large language models (MLLMs) to operate computers and complete tasks, have attracted significant attention for their utility and versatility. A major challenge in developing these agents is collecting large-scale, high-quality trajectories. The standard approach generates synthetic data through a self-improving loop: an agent is placed in a verifiable environment and iteratively fine-tuned on its successful trajectories. Despite its effectiveness, this paradigm exploits only successful trajectories and discards the failed ones, even though failures carry rich information about a model's weaknesses. In this work, we explore a complementary failure-driven self-improvement loop, a data-centric paradigm that turns failed trajectories into agent improvements. Specifically, we employ an LLM to diagnose failure modes, propose inference-time solutions, and generate code patches—lightly verified by humans—that upgrade the agent. We validate this approach with the state-of-the-art OpenCUA-72B model on the OSWorld benchmark, improving the success rate from 42.3% to 48.9%, a gain of 6.6 percentage points, without any additional training cost and with only modest inference overhead. Our results demonstrate that failure-driven self-improvement is a viable complement to success-based pipelines, enabling more efficient agent improvement. Code is available at `https://github.com/snow10072740/Learning_from_Failure`.

1 Introduction

Computer-use agents—systems that employ multimodal large language models (MLLMs) to operate a computer and complete complex tasks such as buying tickets or creating slides—have recently attracted substantial attention for their practical value [1,2,5,8,17,19,22,30,32]. In such settings, an MLLM observes the current computer state (e.g., a screenshot) and predicts the next action conditioned on the task instruction, iteratively producing a trajectory of state–action pairs until the task is completed [26].

A key requirement for building effective computer-use agents is the availability of high-quality trajectories [22]. Although human demonstrations are the most reliable source, they are extremely costly and do not scale. Consequently,

[†]Corresponding authors: `{yuhuiz,syyeung}@stanford.edu` (equal advising).

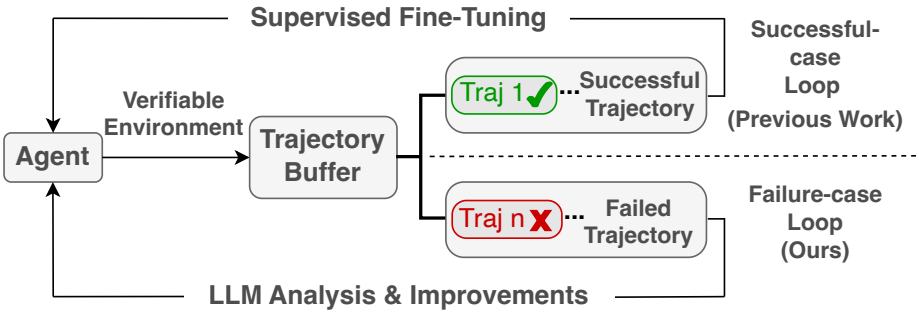


Fig. 1: Illustration of self-improving loops. While prior work focuses on fine-tuning the agent with collected successful trajectories, we explore the failure-case loop, which makes use of the large number of failure cases through LLM analysis and self-improvement.

recent work relies heavily on synthetic trajectories generated in verifiable environments: an agent is placed in an environment, executes a trajectory, and an automatic verifier determines whether the trajectory is correct. Successful trajectories are added to a dataset that is used to further fine-tune the agent, forming the iterative cycle agent → environment → successful trajectory → SFT → improved agent [17, 22, 31]. We refer to this as the *successful-case loop*, which substantially reduces reliance on expensive human annotations.

However, a major limitation of this loop is that all failed trajectories are discarded, despite the substantial cost of building verifiable environments. Although constructing verifiable environments is cheaper than collecting human trajectories, it still requires considerable human engineering effort. This raises a natural question: can we extract value from failed trajectories instead of wasting them?

In this work, we investigate how to leverage failed trajectories to help agents self-improve. We introduce a complementary loop that enhances the agent at inference time, as shown in Figure 1. Given the environment and a collection of failed trajectories, we ask an LLM to analyze these failures, categorize common failure modes, and propose actionable solutions. We then convert these solutions into patch-like code modifications that can be injected into the trained agent's inference-time behavior, yielding an improved agent. This produces a second loop, agent → environment → failed trajectories → LLM analysis & improvement → new agent, which we call the *failure-case loop*. By delivering structured improvements as code-level updates, this loop gives agents a practical mechanism to continuously refine their behavior as they interact with the environment.

We explore this failure-case loop in OSWorld, the most widely used verifiable environment for computer-use agents. Starting from the strongest open-source model, OpenCUA-72B [22], which initially achieves 42.3% accuracy on OSWorld, our LLM-based analysis identifies four major categories of failure: grounding errors, competency gaps, knowledge deficiencies, and redundant loops. For each

category, the LLM proposes a corresponding solution: visual search, terminal execution, knowledge support, and repetition warnings. A human then selects a target solution, minimally adjusts the patch code generated by the LLM, and integrates it into the agent's workflow, which is then re-evaluated on OSWorld (see Appendix). The enhanced agent reaches 48.9% accuracy—an absolute improvement of +6.6 points at no additional training cost and only modest inference overhead. This gain directly expands the proportion of usable environments that can contribute high-quality trajectories to the successful-case loop. A further generalization study shows consistent improvements across different graphical user interface (GUI) benchmarks, indicating that our identified failure categories capture general patterns of GUI-agent deficiencies. Moreover, we observe consistent gains across agents of different families and scales, with stronger models benefiting the most—highlighting the potential of our framework to further amplify the performance of increasingly capable computer-use agents.

Our main contribution is three-fold: (1) we identify a fundamental inefficiency in current agent-improvement pipelines, namely the under-utilization of failed trajectories in success-driven learning loops; (2) we propose a complementary failure-driven self-improvement loop, a data-centric paradigm that leverages failed trajectories to derive inference-time behavioral improvements through LLM-based analysis and lightweight human verification; and (3) we provide, to our knowledge, the first empirical evidence that such a loop can meaningfully improve computer-use agents, demonstrating a 6.6-point absolute gain on OSWorld. Together, our results suggest that failed trajectories are not merely noise but a structured source of supervision, and that incorporating them into an LLM-guided improvement process offers a practical path toward more efficient and continually improving computer-use agents.

2 Related Work

Computer-use agents. Computer-use agents, driven by multimodal large language models (MLLMs), automate graphical user interface (GUI) interactions and real-world tasks [1, 2, 5, 8, 17, 19, 22, 30, 32]. These agents process instructions and visual states (e.g., screenshots) to generate actions such as mouse clicks, keyboard inputs, or commands, enabling navigation in computer environments. Early systems such as Agent S [1] and its successor Agent S2 [2] leverage powerful MLLMs as planners and controllers, introducing hierarchical planning, grounding specialists, and task decomposition to handle long-horizon GUI tasks. More recent work develops native agents for efficient multi-step GUI interaction, such as UI-TARS [17], OpenCUA [22], and Surfer 2 [32], which use a single agent to perform both planning and control. Benchmarks like OSWorld [26] offer verifiable, open-ended tasks for evaluating performance in areas including file management, web browsing, and software operation. In this work, we leverage the state-of-the-art open-source model OpenCUA [22] to explore inference-time self-improvement of a computer-use agent on the widely used OSWorld benchmark [26].

Self-improvement of computer-use agents. Self-improvement enables AI agents to iteratively refine their behavior while minimizing dependence on human annotations [11]. Most existing self-improvement strategies rely on success-driven self-training loops. In frameworks such as OpenCUA, UI-TARS, and Mobile-Agent [17, 22, 31], the agent executes tasks, retains only successful trajectories, and periodically fine-tunes on this curated dataset. This paradigm reduces the need for human demonstrations but discards the majority of experience—namely, failed trajectories—even though they contain valuable diagnostic signals. Alternative approaches include self-evolving agents such as SEAgent [20], SEA [10], and UI-Genie [36], which learn from experience through trial-and-error or iterative boosting. Other methods employ self-evolutionary reinforcement learning for visual grounding [33] or efficient training with reduced data reliance [7]. Our work introduces a complementary failure-driven loop. Instead of relying on additional training, we use an LLM to analyze failed trajectories at inference time, extract systematic failure modes, and apply targeted code or strategy patches that immediately modify the agent's behavior. This yields cost-effective enhancements that seamlessly complement existing success-driven self-training loops.

Error analysis of computer-use agents. Although recent computer-use agents have made promising progress toward general-purpose computer use, several fundamental bottlenecks remain [9, 34]. GUI environments are visually and structurally complex, containing text, icons, widgets, and hierarchical layouts that vary across applications. Prior work highlights the difficulty of building reliable multimodal representations for screens [21,23], while recent GUI-focused grounding studies show that incorrect or incomplete perception is a leading failure mode [15, 24, 35]. Moreover, many GUI tasks demand multi-step workflows and contingent decision-making over evolving interface states; recent web and desktop benchmarks show that reasoning errors grow with trajectory length and that failure recovery remains limited [4, 27, 37]. In this work, we use an LLM to analyze failed trajectories of computer-use agents and identify representative failure modes, producing findings comparable to—and in some cases beyond—those of previous studies. We further use the LLM to propose inference-time corrections for these errors.

3 Method

In this section, we elaborate on the **failure-case loop**, a data-centric self-improvement mechanism that leverages failed trajectories collected during task execution, categorizes systematic failure modes, and synthesizes targeted corrective strategies to address them. The framework organizes failed trajectories into structured, reusable behavioral improvements, enabling the agent to iteratively refine its execution at inference time. We then present a systematic analysis of the prevalent failure patterns observed in GUI-agent interactions, together with a principled and generalizable solution for each.

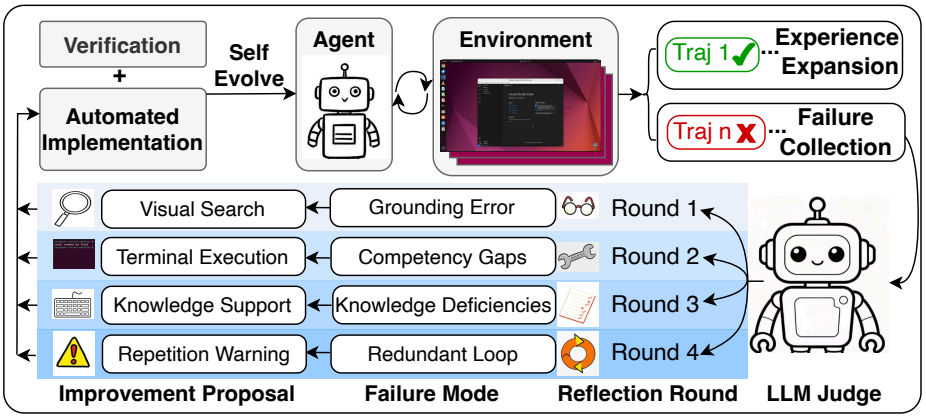


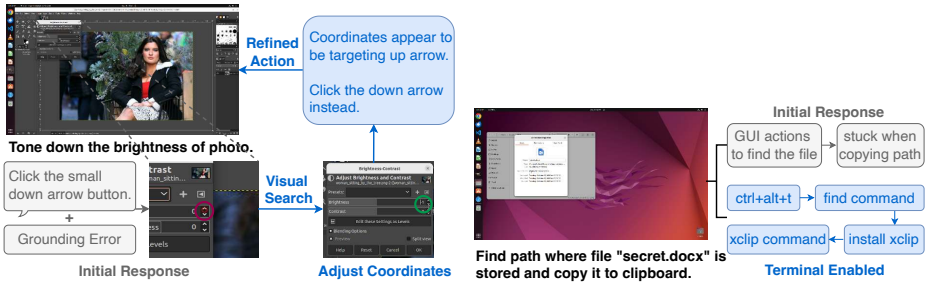
Fig. 2: Overview of our failure-case loop, a self-evolving framework. In each round, failed trajectories are collected through agent rollout; a large language model (LLM) then acts as a meta-controller that diagnoses failure modes, proposes inference-time solutions, and applies code patches. The recipe for OSWorld is generated over multiple rounds of this failure-case loop.

3.1 Framework Overview

We introduce a self-evolving framework, illustrated in Figure 2, that systematically leverages failed trajectories observed during execution. The LLM-guided loop identifies and categorizes failure modes and synthesizes targeted improvement strategies to mitigate them. Over time, these strategies accumulate into a continually updated *set of optimization strategies*, enabling the agent to refine its execution in a data-driven manner. This cycle is computationally efficient and progressively enhances the agent's inference-time capabilities by making effective use of large numbers of failed trajectories.

Table 1: Ablation study of the proposed inference-time optimization strategies. Each strategy improves the inference-time performance of the OpenCUA-72B model on the OSWorld small set, and combining all strategies further boosts performance to 52.74%.

Method	Performance
Baseline:	
OpenCUA-72B (30 steps)	41.67
+ Visual Search	47.22
+ Repetition Detection	44.40
+ Terminal Execution	47.19
+ Knowledge Support	44.44
+ Full Method (Ours)	**52.74**



(a) Visual search strategy. The agent initially mis-localizes the down-arrow in the settings panel; once it is offered a zoomed-in view, it correctly refines its grounding.

(b) Terminal execution strategy. Whereas searching for a file through GUI actions is complicated, three command executions complete the task successfully.

Fig. 3: Case studies of the visual search and terminal execution strategies. In both cases, our framework guides the agent to correctly improve its behavior.

Failure Experience Collection. Within the given verifiable environment, the agent performs rollouts to generate a diverse set of execution trajectories. Each trajectory is evaluated by the environment's built-in reward function, which provides quantitative feedback reflecting task-specific performance. Failed trajectories are preserved as the basis for subsequent diagnosis and improvement.

LLM-Guided Self-Improvement Loop. To extract experience from failed trajectories, we invoke a large language model to conduct targeted error analysis. Each problematic trajectory is fed into the LLM together with its *instruction*, *action history*, and *thought process*, allowing the model to reason about failure modes in context. The LLM then performs a structured diagnosis, identifying underlying bottlenecks such as grounding deficiencies, action redundancy, and operational competency gaps. Based on this analysis, it proposes inference-time improvements, including conceptual adjustments and direct `code implementations` that are *lightly verified by humans*, to enhance the agent's behavior. The LLM introduces four core strategies iteratively, selecting one at each round according to the dominant failure mode, and these strategies accumulate across rounds. After applying an adjustment, the agent executes new rollouts, producing updated failure cases that are fed back into the same diagnostic loop, thereby enabling a continuous refinement cycle. A detailed explanation is provided in Appendix.

Through several rounds of the improvement loop, our framework identifies a set of recurring failure patterns that hinder the agent's progress in complex multimodal environments. These deficiencies reveal fundamental limitations in grounding, control, knowledge, and procedural efficiency.

3.2 Grounding Errors

Failure Mode Analysis. In the first few rounds of LLM-driven failure analysis, grounding errors emerge as one of the most common bottlenecks for GUI agents.

Understanding interface layouts, locating functional elements, and estimating visual coordinates remain particularly challenging, especially in high-resolution or visually cluttered settings. As a result, the agent often misinterprets visual cues or fails to align its actions with the intended targets, leading to inaccurate operations and suboptimal task outcomes.

Proposed Strategy: Visual Search. To improve grounding accuracy, we introduce a **visual search mechanism** that enables multi-round localization and self-refinement for grounding-related actions. Unlike prior GUI grounding approaches that primarily emphasize perception or pre-action grounding, our method integrates visual search as a *first-class module for post-action visual self-verification.*

Specifically, whenever the agent executes a spatially grounded operation such as `click`, `moveto`, or `dragto`, the framework extracts a fixed-size region centered at the target location and generates a zoomed-in view to facilitate fine-grained visual interpretation and spatial reasoning. This localized view is augmented with a visual hint, a red circular marker indicating the original action position. The agent is then prompted to verify its action conditioned on the cropped screenshot, task instruction, action history, and ongoing reasoning context, and to revise the grounding decision if inconsistencies are detected. A more detailed explanation is provided in Appendix.

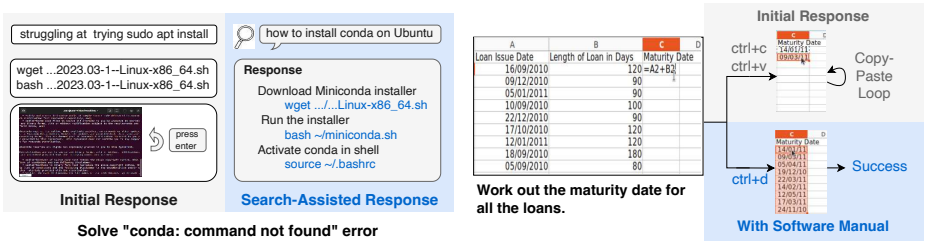
By explicitly closing the loop between action execution and visual verification, this interactive refinement process substantially improves the precision and reliability of grounding in high-resolution, cluttered graphical user interfaces.

Outcome. On the OSWorld small set, visual search improves performance from 41.67 to 47.22, as shown in Table 1. Consider the case in Figure 3a: the agent is asked to tone down the brightness of a photo in GIMP. When it first triggers a grounding error (clicking the up-arrow while attempting to lower the brightness), our framework provides a zoomed-in view together with a visual prompt. The agent then reflects on its action and proposes a corrected coordinate, leading to task success.

3.3 Competency Gaps

Failure Mode Analysis. Another recurring failure mode identified by the LLM is the agent's competency gaps. Although operating systems provide reliable, high-level control interfaces—most notably terminals, which allow direct execution of precise commands—current GUI agents rarely leverage these capabilities, defaulting instead to low-level mouse- or cursor-based manipulation. This reliance on fine-grained interface interaction increases execution volatility and compounds error accumulation, hindering agent performance.

Proposed Strategy: Terminal Execution. Accordingly, the LLM proposes terminal use, activated through a dedicated hotkey, to perform system-level operations. This capability allows the agent to translate operation-intensive tasks into concise command-line instructions, bypassing complex graphical interactions and



(a) Knowledge support via search engine. The agent is unfamiliar with the commands to install conda, but querying GPT-5-mini helps it solve the task.

(b) Knowledge support via software manual. Given LibreOffice hotkeys, the agent bypasses a complex copy–paste procedure and fills all cells with a single hotkey.

Fig. 4: Case studies of the knowledge support strategy. In both cases, our framework guides the agent to correctly improve its behavior.

accomplishing many workflows with greater accuracy and efficiency. A more detailed explanation of this strategy is provided in Appendix.

Outcome. In the case shown in Figure 3b, the agent is asked to locate a specific file and copy its path to the clipboard. Terminal execution provides a direct way to complete this task; otherwise the agent would attempt it through GUI actions and get stuck when copying the path. This strategy improves performance on the OSWorld small set by 5.52 points.

3.4 Knowledge Deficiencies

Failure Mode Analysis. During agent execution and iterative evolution, the LLM identifies gaps in the agent's initial knowledge as a major source of failures, manifesting in patterns such as insufficient multi-step planning and repeated action loops. These issues arise when task-relevant information falls outside the agent's internal context, particularly for domain-specific operations such as application-level procedures or command-line syntax. Lacking access to external references, the agent cannot reliably execute tasks that depend on specialized or factual knowledge.

Proposed Strategy: Knowledge Support. To mitigate knowledge deficiencies, we implement two complementary mechanisms: a **software-manual retriever** that accesses a curated repository of application manuals and hotkey references (see Appendix), and a **search-engine interface** that fetches relevant information beyond the model's internal context window (see Appendix). Together, these components enable the agent to acquire software-level operational knowledge and to autonomously query external knowledge bases when uncertainty arises, improving its capacity to execute domain-specific and knowledge-intensive tasks.

Outcome. Knowledge support improves the performance of the OpenCUA-72B agent from 41.67 to 44.44. An example appears in Figure 4a, where the agent

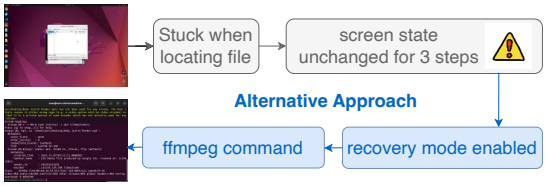


Fig. 5: Repetition-warning strategy. The agent initially gets stuck at a screen state; after three rounds of unproductive attempts, recovery mode is triggered, prompting the agent to adopt an alternative strategy and succeed.

is asked to resolve a "conda: command not found" error. Although the agent initially does not know how to do this, by retrieving a result from GPT-5-mini it learns the correct commands to install conda.

A second example is the LibreOffice case in Figure 4b, where the agent must compute a maturity date for every row of data. While it initially struggles with a `Ctrl+C`/`Ctrl+V` loop, after being given the software manual it learns to use `Ctrl+D` to fill the selected range instantly, completing the task efficiently.

3.5 Redundant Loops

Failure Mode Analysis. The most common failure mode identified by the LLM is redundant action loops, which usually arise when the agent fails to recognize that it is stuck in repetitive or ineffective behavior. For example, after an operation fails, the agent may repeatedly execute the same action sequence without reflecting on the underlying cause or considering alternatives, effectively falling into an action loop. In other cases, the agent performs a sequence of clicks or UI interactions but does not verify whether the expected state transition has actually occurred. As a result, it continues its current plan even when the interface remains unchanged, leading to prolonged stagnation and inefficient exploration. This lack of self-awareness not only wastes many inference steps but also prevents timely recovery from task deviations, ultimately reducing task-completion reliability and overall success rates.

Proposed Strategy: Repetition Warnings. To mitigate redundant action loops, the LLM introduces a systematic **repetition-detection mechanism** that monitors recent interaction history within a fixed-size sliding window of the agent's *thought*, *action*, and *screen-state* traces. Rather than requiring strictly consecutive repetitions, the mechanism examines the *frequency and recurrence patterns* within the window, flagging repetition once occurrences exceed a predefined threshold, which signals stagnation. Specifically:

– **Thought repetition.** We detect repetition when the same or semantically equivalent low-level planning statements reappear multiple times within the

window, indicating that the agent's reasoning loop is not progressing toward a new decision.
- **Action repetition.** If identical action commands (e.g., the same `pyautogui` interaction) occur frequently across the window, we classify this as action repetition, suggesting that the agent is executing without adapting to task feedback.
- **Screen-state repetition.** We encode each screen state into a compressed representation capturing structural and textual cues. If the encoded states remain unchanged across the window despite multiple actions, we detect screen-state repetition, signaling that the agent's actions are not producing the intended interface progress.

When any of these patterns is detected, the mechanism issues an explicit warning to the agent and activates a **recovery mode**. In this mode, the agent is encouraged to reformulate its plan or switch to an alternative interaction strategy, such as exploring a different UI pathway or invoking an auxiliary tool (e.g., a search or system-level command interface). This intervention prevents prolonged stagnation, reduces wasted inference steps, and substantially improves task-completion reliability. A more detailed description of the repetition rule and intervention is provided in Appendix.

Outcome. Table 1 shows the agent's performance before and after the repetition-detection mechanism is applied; it improves performance by 2.73 points. An example is shown in Figure 5: the agent normally gets stuck halfway, but after detecting that the screen state remains unchanged for 3 rounds, it switches to a command-line approach to finish the task.

3.6 Overall Effectiveness

Outcome. Table 1 reports an ablation study of each improvement strategy introduced above, as well as the overall effect when all of them are combined. The OpenCUA-72B model is evaluated on the OSWorld small set—a lightweight subset of OSWorld used for rapid iteration—with a 30-step task limit, in contrast to the full-set, 100-step evaluation reported in Table 2.

Integrating all of these techniques into a unified framework yields a substantial performance boost on the OSWorld small set—one that clearly exceeds what any single technique achieves on its own. Each component addresses a different aspect of the agent's limitations, and their combined use enables the system to operate with greater robustness, adaptability, and efficiency. Overall, the OpenCUA-72B score increases from 41.67 to 52.74, highlighting both the contribution of each individual technique and the benefit of combining their complementary strengths.

Failure Mode Transition. As summarized in Figure 6, we observe a clear shift in the distribution of failure modes, as categorized by the LLM, before and after applying our framework. Because a single failed trajectory may exhibit more than

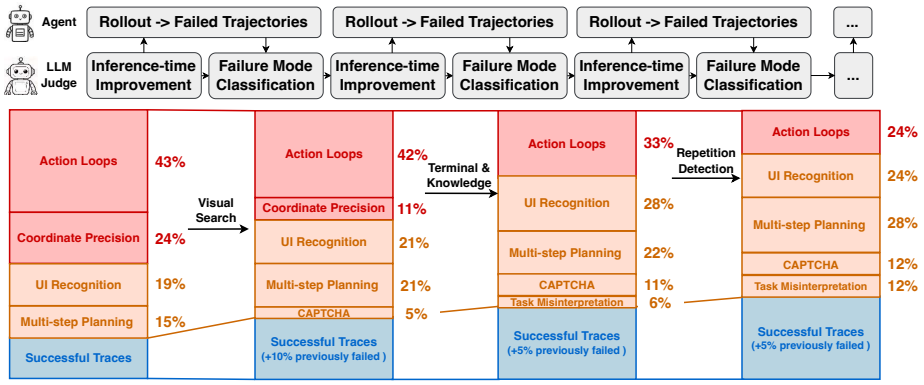


Fig. 6: Failure-mode distribution. The agent's initial failures mainly consist of grounding errors, redundant action loops, and a lack of recovery; our self-evolving framework helps the agent overcome these deficiencies. The remaining failure modes span task comprehension, reasoning consistency, and visual understanding, reflecting higher-level cognitive aspects of agent ability. (Failure-mode proportions are approximately proportional to bar heights; the "success" portion is illustrative only and not to scale.)

one failure mode, these proportions are not mutually exclusive and need not sum to 100%. In the earlier version, failures were dominated by action loops (43%) and coordinate-precision errors (24%), indicating that the agent frequently struggled with precise grounding, knowledge deficiency, and self-recovery. UI-recognition errors (19%) and insufficient multi-step planning (15%) further reflected limitations in perceptual grounding and multi-step reasoning.

After incorporating our improvements, these issues are largely mitigated: the proportion of failed trajectories declines, and the remaining failures shift toward higher-level cognitive aspects of agent ability. After four rounds of our failure-case loop, the remaining failure modes consist of task misinterpretation (12%), CAPTCHA failures (12%), UI-recognition errors (24%), insufficient multi-step planning (28%), and action loops (24%).

This transition demonstrates that our framework effectively resolves the majority of low-level bottlenecks, such as repetitive action loops, grounding errors, and the lack of a self-recovery mechanism. The remaining challenges shift toward higher-level cognitive aspects—task comprehension, reasoning consistency, and visual understanding—revealing deeper limitations in the model's semantic and reasoning capabilities.

4 Results

This section evaluates our failure-case loop on realistic computer-use tasks. In short, it raises OpenCUA-72B from 42.3 to 48.9 on OSWorld (+6.6 points, +15.6% relative) at no additional training cost, our ablation shows each strategy contributes and combining them works best, and the same loop transfers across

model scales and four heterogeneous benchmarks without retraining. We detail the setup, main results, ablations, and generalization studies below.

4.1 Experimental Setup

We design a comprehensive evaluation to test whether our *failure-case loop* enhances agent performance in realistic operating environments. The experiments are conducted on a complex computer-use benchmark, with systematic baseline comparisons, detailed ablation analysis, and generalization tests.

Benchmark. We validate our framework on the **OSWorld** [26] benchmark, chosen for its real-world complexity and broad task coverage across diverse operating-system scenarios such as document editing and file management. This environment provides a high-fidelity testbed for evaluating both the agent's *grounding accuracy* and *operational competence* under realistic conditions.

Baseline. As the reference model, we employ the open-source **OpenCUA-72B** [22], which has strong general reasoning capabilities and partial familiarity with OSWorld's action space. Its balanced performance across general and system-level tasks makes it a suitable baseline for evaluating the improvements brought by our failure-driven self-improvement framework.

Failure-Case Loop. In each round, failed-trajectory collection, trajectory analysis, failure diagnosis, and code refinement are conducted by **Claude 4.5 Sonnet**, which serves as a meta-level reasoner. It analyzes erroneous trajectories, identifies behavioral bottlenecks, and proposes targeted improvements that are incorporated into the agent's inference-time optimization loop with light human verification. In practice, the human role is limited to lightweight verification and candidate selection: over 97% of LLM-generated refinements are accepted without modification, indicating that the loop is predominantly model-driven. We run the loop for four rounds, identifying failure modes and proposing inference-time solutions as discussed in Section 3.

Meta-Controller. We select the meta-controller from among Claude 4.5 Sonnet, GPT-5.2, Gemini 3 Flash, and the open-source Qwen3-VL-32B-Instruct by comparing their performance. While GPT-5.2 and Qwen3-VL-32B-Instruct identify certain failure modes, their analyses are less comprehensive and less consistent in proposing implementable refinements, and Gemini 3 Flash exhibits comparatively weaker code-level implementation ability. Since the meta-controller must jointly analyze failures, propose actionable improvements, and implement code modifications, Claude 4.5 Sonnet demonstrates the strongest overall performance and is therefore adopted in our framework.

Evaluation and Ablation. We evaluate the refined framework on OSWorld and compare it against both the baseline and other high-performing systems on the leaderboard. To ensure a fair comparison, our framework retains exactly the same

tool access, environment configurations, and action spaces as the baseline agents. To quantify the contribution of each component, Section 3.6 reports ablations on the OSWorld small set that isolate **visual search**, **terminal execution**, **knowledge support**, and **repetition warnings**, measuring their individual effects on **task success rate**. These analyses show that the integrated system outperforms any individual module, combining their complementary strengths into a single, more robust agent.

Cross-Model Generalization. To assess the breadth and robustness of our method's generalization, we apply the same failure-driven self-improvement loop to models of different capacities and origins—most notably **OpenCUA-32B** [22] and **GUI-Owl-32B** [31]. These two models differ substantially in architecture, training philosophy, and pretraining sources, providing a meaningful testbed for the transferability of our framework across heterogeneous backbones. We report their performance on OSWorld with the improvement strategies of Section 3 applied.

Cross-Benchmark Generalization. To evaluate cross-benchmark generalization, we directly transfer the failure-derived patches mined from OSWorld to heterogeneous benchmarks: **OmniACT** [12] for desktop-level agent tasks, **AndroidControl** [14] for mobile environments, **ScreenSpotPro** [13] for advanced GUI grounding, and **WebVoyager** [6] for web-based multimodal interaction. We use **Qwen3-VL-32B-Instruct** as the default backbone for these tests under identical evaluation protocols.

4.2 Results

Main Results. Our primary evaluation uses the **OpenCUA-72B** model as the backbone for our failure-case loop, with performance measured on the **OSWorld** benchmark over 100 steps.

As shown in Table 2, applying our framework improves performance from **42.3** to **48.9**, an absolute gain of **+6.6** points (**+15.6%** relative). This highlights the synergy between a strong base model and our failure-driven self-improvement mechanism, which progressively refines reasoning, grounding, and recovery throughout interaction.

Importantly, this improvement is achieved without any additional training cost, since all enhancements are applied at inference time. The system introduces only modest computational overhead—about an 8% increase in runtime—while reducing interaction steps by roughly 15%, indicating improved execution efficiency.

Generalization Across Models, Scales, and Environments. To verify the generalization of our approach, we apply the same failure-driven self-improvement framework to models of different capacities and origins, specifically **OpenCUA-32B** and **GUI-Owl-32B**. As reported in Table 2, across both settings we observe consistent **+10–12%** relative gains regardless of model size or provenance.

Table 2: Comparison with state-of-the-art methods on the OSWorld [26] benchmark. The second column reports the number of steps used by each agent, where one step is a single agent–environment interaction (action + observation), and success rate (%) is the evaluation metric. For our framework, the main results are the mean ± standard deviation across 3 independent runs; * denotes configurations evaluated in a single run due to resource constraints.

Agent Model	**Step**	**Success Rate**
Proprietary Models		
Claude 3.7 Sonnet [3]	100	28.0
OpenAI CUA 4o [16]	200	38.1
UI-TARS-1.5 [18]	100	42.5
OpenAI CUA o3 [16]	200	42.9
Open-Source Models		
Aria-UI w/ GPT-4o [29]	15	15.2
Aguvis-72B w/ GPT-4o [28]	15	17.0
UI-TARS-72B-SFT [18]	50	18.8
Agent S w/ Claude-3.5-Sonnet [1]	15	20.5
Agent S w/ GPT-4o [1]	15	20.6
UI-TARS-72B-DPO [18]	15	22.7
UI-TARS-72B-DPO [18]	50	24.6
UI-TARS-1.5-7B [18]	100	26.9
Jedi-7B w/ GPT-4o [25]	100	27.0
Agent S2 w/ Claude-3.7-Sonnet [2]	50	34.5
Agent S2 w/ Gemini-2.5-Pro [2]	50	41.4
GUI-Owl-32B [31]	100	19.0*
+ Ours	100	**21.3***
OpenCUA-32B [22]	100	34.5*
+ Ours	100	**38.2***
OpenCUA-72B [22]	100	42.3 ± 2.6
+ **Ours (Main)**	100	$\mathbf{48.9 \pm 1.2}$

These uniform improvements indicate that the proposed recipe generalizes reliably across scales and sources, reinforcing the view that our framework acts as a scalable, model-agnostic amplifier of base-model capabilities.

For cross-benchmark generalization, as shown in Table 3, incorporating failure-derived patches mined from OSWorld yields consistent improvements across all evaluated benchmarks. The largest gains appear on WebVoyager (+4.10 points) and AndroidControl (+7.86 points), both of which involve long-horizon planning and interactive command execution. This further supports the hypothesis that failure trajectories mined from OSWorld capture generalizable error patterns rather than environment-specific heuristics.

5 Discussion

This work presents a self-evolving framework centered on a structured **failure-case loop** for GUI agents, which leverages failed trajectories at inference time

Table 3: Cross-benchmark generalization results. The results show that our method captures transferable failure patterns and outperforms the base agents. All experiments are repeated three times except for WebVoyager, and we report the mean and standard deviation.

	OmniACT	AndroidControl	ScreenSpotPro	WebVoyager
Base	4.77±0.02	28.37±0.13	27.50±0.35	23.80
Ours	**6.90±0.10**	**36.23±0.22**	**30.74±0.27**	**27.90**

instead of discarding them. Rather than treating errors as terminal outcomes, the framework organizes them into actionable signals—identifying bottlenecks in grounding, control, knowledge usage, and procedural efficiency—and applies targeted inference-time strategies to mitigate them in a semi-automatic manner.

The resulting plug-and-play recipe operates entirely at inference time, requiring no additional training or data collection, yet consistently strengthens agent performance on OSWorld, a highly challenging computer-use environment. Applied to a strong baseline such as **OpenCUA-72B**, the framework improves performance from **42.3** to **48.9**, with similarly robust **+10–12%** relative gains across models of different origins and scales, including **OpenCUA-32B** and **GUI-Owl-32B**. It also generalizes well to other GUI-agent benchmarks.

This failure-case loop complements existing success-based loops, jointly improving trajectory quality by amplifying successful behaviors and correcting failures. Importantly, we show that substantial improvements can be achieved at inference time through structured interventions, without any weight updates.

The framework further scales through a sequential, failure-driven update process in which each iteration targets a newly identified failure mode rather than accumulating ad-hoc patches. As shown in our experiments, stacking four structured updates consistently outperforms any single update, indicating that improvements compose constructively and scale reliably.

Moreover, learned reward models trained on prior interaction data could be integrated into our framework to automatically identify erroneous or suboptimal trajectories without human feedback, further improving scalability and enabling continuous self-refinement in open-ended environments.

Overall, these results demonstrate a **scalable, general, and data-efficient** route for advancing GUI agents. By systematically leveraging failure cases as continual inference-time signals, the framework enables agents to extract substantially more value from their interactions with the environment, paving the way toward more efficient and self-improving systems.

Acknowledgments. We gratefully acknowledge Lambda, Inc. for providing partial computational support for this project. S.Y. is a Chan Zuckerberg Biohub — San Francisco Investigator.

References

1. Agashe, S., Han, J., Gan, S., Yang, J., Li, A., Wang, X.E.: Agent s: An open agentic framework that uses computers like a human. arXiv preprint arXiv:2410.08164 (2024)
2. Agashe, S., Wong, K., Tu, V., Yang, J., Li, A., Wang, X.E.: Agent s2: A compositional generalist-specialist framework for computer use agents. arXiv preprint arXiv:2504.00906 (2025)
3. Anthropic: Claude 3.7 sonnet and claude code. Technical report, Anthropic (2025), https://www.anthropic.com/news/claude-3-7-sonnet, system Card
4. Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., Su, Y.: Mind2web: Towards a generalist agent for the web (2023), https://arxiv.org/abs/2306.06070
5. Feizi, A.: Grounding computer use agents on human demonstrations. arXiv **2511.07332** (Nov 2025). https://doi.org/10.48550/arXiv.2511.07332, https://arxiv.org/abs/2511.07332, v1
6. He, H., Yao, W., Ma, K., Yu, W., Dai, Y., Zhang, H., Lan, Z., Yu, D.: Webvoyager: Building an end-to-end web agent with large multimodal models (2024), https://arxiv.org/abs/2401.13919
7. He, Y.: Efficient agent training for computer use. arXiv preprint arXiv:2505.13909 (2025). https://doi.org/10.48550/arXiv.2505.13909, arXiv:2505.13909v1
8. Hu, X.: Os agents: A survey on mllm-based agents for general computing devices use. arXiv preprint arXiv:2508.04482 (2025), https://doi.org/10.48550/arXiv.2508.04482, accepted by ACL 2025 (Oral)
9. Hu, X., Xiong, T., Yi, B., Wei, Z., Xiao, R., Chen, Y., Ye, J., Tao, M., Zhou, X., Zhao, Z., Li, Y., Xu, S., Wang, S., Xu, X., Qiao, S., Wang, Z., Kuang, K., Zeng, T., Wang, L., Li, J., Jiang, Y.E., Zhou, W., Wang, G., Yin, K., Zhao, Z., Yang, H., Wu, F., Zhang, S., Wu, F.: OS agents: A survey on MLLM-based agents for computer, phone and browser use. In: Che, W., Nabende, J., Shutova, E., Pilehvar, M.T. (eds.) Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 7436–7465. Association for Computational Linguistics, Vienna, Austria (Jul 2025). https://doi.org/10.18653/v1/2025.acl-long.369, https://aclanthology.org/2025.acl-long.369
10. Huo, Y.: Sea: Self-evolution agent with step-wise reward for computer use. arXiv (Aug 2025), https://arxiv.org/abs/2508.04037, arXiv:2508.04037 [cs.AI]
11. Juan, X.: A survey of self-evolving agents: On path to artificial super intelligence. arXiv **2507.21046** (2025). https://doi.org/10.48550/arXiv.2507.21046, v3
12. Kapoor, R., Butala, Y.P., Russak, M., Koh, J.Y., Kamble, K., Alshikh, W., Salakhutdinov, R.: Omniact: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web (2024), https://arxiv.org/abs/2402.17553
13. Li, K., Meng, Z., Lin, H., Luo, Z., Tian, Y., Ma, J., Huang, Z., Chua, T.S.: Screenspot-pro: Gui grounding for professional high-resolution computer use (2025), https://arxiv.org/abs/2504.07981
14. Li, W., Bishop, W., Li, A., Rawles, C., Campbell-Ajala, F., Tyamagundlu, D., Riva, O.: On the effects of data scale on ui control agents (2024), https://arxiv.org/abs/2406.03679
15. Liu, Y., Li, P., Xie, C., Hu, X., Han, X., Zhang, S., Yang, H., Wu, F.: Infigui-r1: Advancing multimodal gui agents from reactive actors to deliberative reasoners. arXiv preprint arXiv:2504.14239 (2025)

16. OpenAI: Openai o3 and o4-mini system card. Technical report, OpenAI (2025), https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf, system Card
17. Qin, Y., Ye, Y., Fang, J., Wang, H., Liang, S., Tian, S., Zhang, J., Li, J., Li, Y., Huang, S., Zhong, W., Li, K., Yang, J., Miao, Y., Lin, W., Liu, L., Jiang, X., Ma, Q., Li, J., Xiao, X., Cai, K., Li, C., Zheng, Y., Jin, C., Li, C., Zhou, X., Wang, M., Chen, H., Li, Z., Yang, H., Liu, H., Lin, F., Peng, T., Liu, X., Shi, G.: Ui-tars: Pioneering automated gui interaction with native agents (2025), https://arxiv.org/abs/2501.12326
18. Qin, Y., Ye, Y., Fang, J., Wang, H., Liang, S., Tian, S., Zhang, J., Li, J., Li, Y., Huang, S., et al.: Ui-tars: Pioneering automated gui interaction with native agents. arXiv preprint arXiv:2501.12326 (2025)
19. Song, L.: Coact-1: Computer-using agents with coding as actions. arXiv **2508.03923** (2025). https://doi.org/10.48550/arXiv.2508.03923, v2
20. Sun, Z.: Seagent: Self-evolving computer use agent with autonomous learning from experience. arXiv **2508** (2025), https://arxiv.org/abs/2508.04700, arXiv:2508.04700 [cs.AI]
21. Wang, B., Li, G., Zhou, X., Chen, Z., Grossman, T., Li, Y.: Screen2words: Automatic mobile ui summarization with multimodal learning (2021), https://arxiv.org/abs/2108.03353
22. Wang, X., Wang, B., Lu, D., Yang, J., Xie, T., Wang, J., Deng, J., Guo, X., Xu, Y., Wu, C.H., Shen, Z., Li, Z., Li, R., Li, X., Chen, J., Zheng, B., Li, P., Lei, F., Cao, R., Fu, Y., Shin, D., Shin, M., Hu, J., Wang, Y., Chen, J., Ye, Y., Zhang, D., Du, D., Hu, H., Chen, H., Zhou, Z., Yao, H., Chen, Z., Gu, Q., Wang, Y., Wang, H., Yang, D., Zhong, V., Sung, F., Charles, Y., Yang, Z., Yu, T.: Opencua: Open foundations for computer-use agents (2025), https://arxiv.org/abs/2508.09123
23. Wu, J., Zhang, X., Nichols, J., Bigham, J.P.: Screen parsing: Towards reverse engineering of ui models from screenshots. In: The 34th Annual ACM Symposium on User Interface Software and Technology. p. 470–483. UIST '21, ACM (Oct 2021). https://doi.org/10.1145/3472749.3474763, http://dx.doi.org/10.1145/3472749.3474763
24. Wu, Z.: See, think, act: Teaching multimodal agents to effectively interact with gui by identifying toggles (09 2025). https://doi.org/10.48550/arXiv.2509.13615, https://arxiv.org/abs/2509.13615
25. Xie, T., Deng, J., Li, X., Yang, J., Wu, H., Chen, J., Hu, W., Wang, X., Xu, Y., Wang, Z., et al.: Scaling computer-use grounding via user interface decomposition and synthesis. arXiv preprint arXiv:2505.13227 (2025)
26. Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T.J., Cheng, Z., Shin, D., Lei, F., et al.: Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. Advances in Neural Information Processing Systems **37**, 52040–52094 (2024)
27. Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T.J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., Yu, T.: Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments (2024), https://arxiv.org/abs/2404.07972
28. Xu, Y., Wang, Z., Wang, J., Lu, D., Xie, T., Saha, A., Sahoo, D., Yu, T., Xiong, C.: Aguvis: Unified pure vision agents for autonomous gui interaction. arXiv preprint arXiv:2412.04454 (2024)
29. Yang, Y., Wang, Y., Li, D., Luo, Z., Chen, B., Huang, C., Li, J.: Aria-ui: Visual grounding for gui instructions. arXiv preprint arXiv:2412.16256 (2024)

30. Yao, H.: A survey on agentic multimodal large language models. arXiv **2510.10991** (Oct 2025). https://doi.org/10.48550/arXiv.2510.10991, https://arxiv.org/abs/2510.10991
31. Ye, J., Zhang, X., Xu, H., Liu, H., Wang, J., Zhu, Z., Zheng, Z., Gao, F., Cao, J., Lu, Z., Liao, J., Zheng, Q., Huang, F., Zhou, J., Yan, M.: Mobile-agent-v3: Fundamental agents for gui automation (2025), https://arxiv.org/abs/2508.15144
32. Yuan, K.: Surfer 2: The next generation of cross-platform computer use agents. arXiv **2510.19949** (2025). https://doi.org/10.48550/arXiv.2510.19949, https://arxiv.org/abs/2510.19949, v2
33. Yuan, X.: Enhancing visual grounding for gui agents via self-evolutionary reinforcement learning. arXiv **2505** (2025). https://doi.org/10.48550/arXiv.2505.12370, https://arxiv.org/abs/2505.12370, arXiv:2505.12370 [cs.AI]
34. Zhang, C.: Large language model-brained gui agents: A survey. arXiv preprint arXiv:2411.18279 (May 2025). https://doi.org/10.48550/arXiv.2411.18279, https://arxiv.org/abs/2411.18279
35. Zheng, B., Gou, B., Kil, J., Sun, H., Su, Y.: Gpt-4v(ision) is a generalist web agent, if grounded (2024), https://arxiv.org/abs/2401.01614
36. Zhou, A.: Ui-genie: A self-improving approach for iteratively boosting mllm-based mobile gui agents. arXiv (May 2025). https://doi.org/10.48550/arXiv.2505.21496, https://arxiv.org/abs/2505.21496, version 1
37. Zhou, S., Xu, F.F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., Neubig, G.: Webarena: A realistic web environment for building autonomous agents (2024), https://arxiv.org/abs/2307.13854